A Rule Selection Method for Automated Reasoning (Algebraic Systems and Theoretical Computer Science)

AUTHOR(S):

Kobayashi, Hidetune; Ono, Yoko

数理解析研究所講究録
第1809巻 2012年 93-99

A Rule Selection Method for Automated Reasoning

Hidetune Kobayashi (Institute of Computational Logic)
Yoko Ono (Yokohama City University)

1 Introduction

H-prover [1] is an automated reasoning system using formal and logical method to prove propositions on set theory. H-prover is designed to generate proof steps for a given proposition to be proved. A proof is a sequence of propositions to used to rewrite the original proposition to some apparently true propositions, and H-prover generates proofs to some propositions. Therefore it is a key point to choose proper rules to prove a proposition. This is a report to show a method to select proper rules among the rules stored in the data base.

2 Structure of an automated reasoning system "H-prover"

In this section, we explain how "H-prover" consists of, and how it works.

2.1 Parts of H-prover

H-prover is consisting of four parts:

1) Isabelle/HOL [2] as an inference engine

2) ProofGeneral as an interface

3) proof controller

4) a database of mathematical knowledge

ProofGeneral is an interface for interactive proof of a proposition, it is incorporated in emacs. We type in a proposition to be proved into emacs, and throw the proposition into Isabelle by pressing the start button. Then Isabelle returns a proposition interpreted by Isabelle, and ProofGeneral displays the proposition returned from Isabelle. Then we insert a rule with adjusted variables to the proposition and press the start button to ask Isabelle to rewrite the proposition. This time, Isabelle returns some propositions which as a whole is a sufficient condition to show the original proposition to be true. Repeating this procedure, finally we obtain a set of apparently true propositions. A role of H-prover is give a rule with adjusted variables for the given proposition, control a proof procedure and put into the data base the original proposition after proved. The main part of proof controller is written in emacs lisp and it is incorporated in emacs. The rule selection functions are written in C language and plpgsql and they works within postgreSQL server which contains rules and hints for a proof.

2.2 Trees in the Data Base

To store rules in the data base, we express rules in tree structure. By using trees, we can match variables of a rule for variables of a proposition to prove. Here, we give an example of a proposition expressed in tree:

`Isabelle expression` $[|P\ c;\ Q\ c\ |] \Longrightarrow \exists c.\ P\ c \wedge Q\ c$

```
Tree expression (LrarS (lrBRK (P c sclS Q c)) (exS $c dS andS (P $c) (Q $c)))
```

As above, Isabelle expression is converted to tree structure. In postgreSQL, we have to treat such trees, we prepared lisp like functions executed within postgreSQL. In the next subsection, we introduce some lisp like functions briefly. The rules are stored in the table "`propositions`" having columns listed below.

no	thy	kind	locale	name	simp	num_prem	tree
int	text	text	text	text	int	int	text

premises	conclusion	num_cn_vars	num_t_vars
text[ ]	text	int	int

We have another table " prop_to_prove " which store proposition to prove. The table has columns

num_prem1	tree1	premises1	conclusion1	num_cn_vars1	num_t_vars1
Int	text	text[ ]	text	int	int

When we choose rules for a proposition to prove, we use these two tables and make a view which contains candidate rules to apply.

2.3 LISP like functions in postgreSQL

In postgreSQL, type of the tree is declared as text, and fundamental LISP functions car, cdr, cons, nth, append, list, sreverse are written in C language. Here, we note that in postgreSQL, there is already the function named reverse, therefore we named sreverse the function having the same effect as reverse in LISP. In postgreSQL, car works as:

```
> select car('((a b) c)');
   car
-------
 (a b)
(1 row)
```

The function " car " is defined as :

```
create or replace function car(text) returns text as
   '/home/hkb/pgsql/C/lisp', 'car'  language c strict;

PG_FUNCTION_INFO_V1(car);
Datum car(PG_FUNCTION_ARGS)
{
  text  *arg = PG_GETARG_TEXT_P(0);
  int32 new_text_size, a, block_size, arg_size = VARSIZE(arg)-VARHDRSZ;
  char  *VAR, buf[10];
  Params; VAR = VARDATA(arg); Kind_of_arg(a, VAR, arg_size);
  switch (a)
   {case 1:{ NIL_nil(buf, block_size); VAR = buf; break;}
    case 2:{ ERROR_er(buf, block_size); VAR = buf;   break;}
    case 3:{ VAR = VAR+1; arg_size = arg_size - 2;
                  Block_end(VAR, arg_size, block_size);  }
   }
      new_text_size = block_size + VARHDRSZ;
      text *new_text = (text *) palloc(new_text_size);
      SET_VARSIZE(new_text, new_text_size);
      memcpy(VARDATA(new_text), VAR, block_size);
      PG_RETURN_TEXT_P(new_text);
 }
```

In the above source code, Kind_of_arg, NIL_nil, ERROR_er and Block_end are macros.

3 Rough rule selection

Since there are hundreds of rules in the DB, we choose rules in two steps. At first, we select rules from the table "propositions" by using three functions root_char, left_char and right_char. Using these functions, H-prover compares roots of conclusion trees, roots of left children of conclusions and roots of right children of conclusions.

3.1 Characters of root, left-child and right-child

root_char is a function taking root of a tree defined as:

```
create function root_char(tree text) returns text as $$
begin
    if reserved_sym(car(root_of_tree(tree))) = 't' then
        return car(root_of_tree(tree));
    else  return 'L';  end if;
end; $$ language plpgsql;

left_char and right_char are defined similarly
    to take the root of left_child and right_child respectively.
We make a view as

create view selected_rules as select  *  from propositions,
prop_to_prove where root_char(conclusion)= root_char(c_tree)
    and left_char(conclusion)  = left_char(c_tree)
    and right_char(conclusion) = right_char(c_tree)
    and num_cn_vars <= num_cn_vars1;
```

First selection of rules is executed as

1. Give a proposition to prove in ProofGeneral, and throw the proposition into Isabelle.
2. Isabelle returns a proposition in "*goal*" buffer.
3. H-prover takes the proposition from "*goal*", then H-prover converts the proposition to tree and puts the tree into the table "prop_to_prove" by using update command.
4. The view "selected_rules" has rules having the same root_char, left_char, right_char and having less number of variables than the proposition to prove.

Grouping by root_char, left_char, right_char, we have 219 groups. The largest group is the group consisting of 52 rules including iffD2. The average number of members is 2, therefore the first selection step is works fairly well.

name	count
iffD2	52
sym	34
subsetD	12
eta_contract_eq	10
ssubst	10
TOTAL 219 groups	AVG 2/group

Now, we see why there are so many rules in iffD2 group. IffD2 is the rule

$$[|P = Q; Q|] \Rightarrow P$$

which is represented as

```
(LrarS (lrBRK (= (P) (Q) sclS Q)) (P))
```

in tree expression. The conclusion tree of the tree is `(P)`. Since the first selection checks only the conclusion of a tree, `root_char of (P)` is `L`, `left_char` is `nil`, `right_char` is `nil`, there is no way to distinguish this rule from the other rules having simple conclusion.

3.2 Position of variables

A position of variables is a list of depth 2 like
`((Cn L P) (Pr_2 L Q) (Pr_1 =) (Pr_1 l L P) (Pr_1 r L Q))`.
Each inner list is a list of symbols indicating a position and a variable at the position. The header of each inner list is a symbol to specify a premise or the conclusion a variable is belonging to. The symbol L shows a variable is a leaf of a tree. The last element is a variable or an operator:

`(Cn L P)` the variable P is a leaf of the conclusion tree (P),

`(Pr_1 =)` the operator = is the root of the tree of the first premise,

`(Pr_1 l L P)` the variable P is the leaf of the left child of the first premise.

Adding `tree_var_pos to root_char, left_char, right_char` in SQL group by condition, we have the following result.

name	count
ssubst	1
all_dupE	1
rev_notE	1
swap	1
Other group	1

From this result we see that `tree_var_pos` , the function giving porision of variables, separates all rules.

4 Detailed rule selection

Comparing conclusions of a proposition to prove and that of a rule in the view `selected_rules`, we can obtain more detailed data to prove the proposition.

4.1 Compare conclusions

To illustrate the procedure of selection, we give a simple proposition to prove:

$$[|P\ c;\ Q\ c|] \Longrightarrow \exists c.\ P\ c \wedge Q\ c$$

We convert this into tree, and put it into the table `prop_to_prove`. Then we have nine rules as in the table.

exI
ex1_implies_ex
exCI
bexI
rev_bexI
bexCI
UNIV_witness
psubset_imp_ex_mem
mk_disjoint_insert

Comparing conclusions as

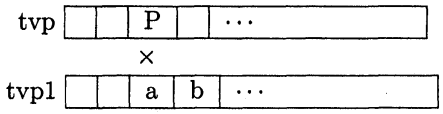
```
select name, compare_conclusions(conclusion, conclusion1)
from selected_rules, prop_to_prove;
```

we obtain the following table.

exI	((P (lmbS xxx1 dS andS (P xxx1) (Q xxx1))))
ex1_implies_ex	((P (lmbS xxx1 dS andS (P xxx1) (Q xxx1))))
exCI	((P (lmbS xxx1 dS andS (P xxx1) (Q xxx1))))
bexI	nil
rev_bexI	nil
bexCI	nil
UNIV_witness	nil
psubset_imp_ex_mem	nil
mk_disjoint_insert	nil

Here (lmbS xxx1 dS andS (P xxx1) (Q xxx1)) is tree expression of "λxxx1. P xxx1 $\wedge$ Q xxx1". The function compare_conclusions is designed as:

```
let tvp be tree_var_pos(conclusion) and let tvp1 be tree_var_pos(conclusion1).
```



If the last element of the data at the first different position is a variable, say P, we cut out the counterpart of P in conclusion1 by using the position data at the different position. If P is in the left child of conclusion tree, we check the similar position in the right child.

In the above table, rules exI, ex1_implies_ex and exCI have the same conclusion. Therefore, we cannot separate these rules only by comparing conclusions.

To test the function works, we try another type of exist tree which is called bex-tree. We give a proposition having bex conclusion:

not_subseteq:" $\neg$ A $\subseteq$B $\Longrightarrow$ a $\in$A. a $\notin$ B"

We have the following table:

exI	nil
ex1_implies_ex	nil
exCI	nil
bexI	((P (lmbS xxx1 dS ninS (xxx1) (B))) (Wh (A)))
rev_bexI	((P (lmbS xxx1 dS ninS (xxx1) (B))) (Wh (A)))
bexCI	((P (lmbS xxx1 dS ninS (xxx1) (B))) (Wh (A)))
UNIV_witness	nil
psubset_imp_ex_mem	nil
mk_disjoint_insert	nil

Now, we compare the rule exI and the rule ex1_implies_ex:

The latter means if there is only one element x satisfying P, then there is an element satisfying P. This is very simple and easy to understand for us, but not so easy for term rewriting by machine. In next section we discuss checking premises, and we note that the above example is an simple example that shows unless checking premises, we cannot decide a solution.

```
exi:              P x ⟹ ∃ x. P x
ex1_implies_ex:   ∃! x. P x ⟹ ∃ x. P x
```

4.2 Compare premises

In this subsection, we discuss a problem concerning with checking premises.

```
Start to prove: b ∈ f ' A ⟹ ∃ a ∃ A. b = f a,
```

In the view "selected_rules", we have nine rules as in the table in subsection 4.1. Comparing conclusion of the rule bexI

```
bexI:  [|P x; x ∃ A|]⟹ ∃ x ∃ A. P x,
```

we obtain λ xxx1. b = f xxx1 as P. However, there is no candidate variable for the local variable x, we cannot apply bexI. A proof of the proposition b ∈ f ' A ⟹ ∃ A. b = f a is obtained as follows:

```
Apply (simp only:image_def)    ---> b ∈ {y. ∈ x ∈ A. y = f x} ⟹ ∃ a ∈ A. b = f a.
Apply (erule CollectE)         ---> ∃ a ∈ A. b = f a ⟹ ∃ a ∈ A. b = f a f
```

Since the prover does not know the definition of the image, in the first line we expand image_def, and in the second line, we give erule CollectE as a method to treat a set. The idea in the second line, we will call it as " mathematical knowledge ". Almost all propositions in mathematics cannot be proved only by simple term rewriting. By virtue of mathematical knowledge, we can make a proof to a mathematical proposition. Therefore how to store mathematical knowledge and how to use stored mathematical knowledge is very important future work.

Now we present a function " check_tree_1 " giving a solution to some simple propositions.

```
create function check_tree_1(tree text, conclusion text, tree1 text, conclusion1 text)
returns text as $$
declare
    vars      text  default 'nil';
    l_vars    text  default 'nil';
    op_pairs  text  default 'nil';
    op_pairs1 text  default 'nil';
    len_1     integer  default  0;
    i         integer  default  0;
    res       text  default 'nil';
    res_1     text  default 'nil';
begin
if     assumption_p(tree1) = 't' then return 'assumption';
elseif ex1_in_prem_equal_concl(tree1, num_premises(tree1)) = 't'
       then return 'ex1_implies_ex';
elseif 'CHECK_premises_1' = 't' then return 'SOME_ACTION_1';
elseif 'CHECK_premises_2' = 't' then return 'SOME_ACTION_2';
else
    res_1 := compare_conclusions(conclusion, conclusion1);
    len_1 := slength(res_1);
    vars  := var_list(tree);
    l_vars:= vars_of_QF_tree(conclusion1); -- later use QFSL-tree
  if l_vars = 'nil' then return list(res_1);
  else
     op_pairs := op_operands(vars, tree_var_pos(tree));
     op_pairs1:= op_operands(var_list(tree1), tree_var_pos(tree1));
     while i < len_1        LOOP
```

```
        res:= append(lcons(nth(i, res_1), make_pairs(
        list (operand_of (car(nth(i, res_1)), pm_op_pairs(op_pairs)),
              operand_of (car(nth(i, res_1)), pm_op_pairs(op_pairs1))))),
                                                                  res);
        i := i + 1;
    END LOOP;
  end if;
   return res;
end if;
end; $$ language plpgsql;
```

This is a first version to obtain a solution. It consists of two parts:

1) simple checks in two points only:

   a, is there a same tree as conclusion in premises,

   b, is there an expression as the premise in `ex1_implies_ex`.

2) make pair of related variables one from a rule and another from `prop_to_prove`.

For the proposition `[|P c; Q c|]` $\Longrightarrow$ `∃ c. P c ∧ Q c`, by using this "`check_tree_1`", the prover gives automatically a sequence of proof steps as

```
lemma ex_conjI:" [|P c; Q c |] ⟹ ∃ c. P c ∧  Q c"
 apply (rule_tac P = "λ xxx1. P xxx1 ∧ Q xxx1" and x = "c" in exI)
 apply (rule conjI)
apply assumption+
done
```

In addition to the proof method "`rule_tac`", there are proof methods "`drule`", "`erule`" and "`frule`" which modify premises. A work to write functions realizing those methods is our urgent task.

Acknowledgement

The idea of this report is an outgrowth of a study in the key laboratory of mathematics mechanization of Chinese academy of sciences. We are grateful to the institute for giving us nice circumstance for study.

References

[1] H. Kobayashi and Y. Ono, An Application of the Formal Method to Statistics, Proceedings of the 2009 International Symposium on Computing, Communication and Control,238- 241, 2009.

[2] T. Nipkow et.al., Isabelle/HOL: A Proof Assistant for Higher- Order Logic. Springer, 2002.